A dynamic dictionary stores \((key, value)\)-pairs and supports:

- \(\text{add}(key, value)\), \(\text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by chaining

\(n\) arbitrary operations arrive online, one at a time.

A set \(H\) of hash functions is weakly universal if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr(h(x) = h(y)) \leq \frac{1}{m}
\]

\((h\) is picked uniformly at random from \(H\))

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.
A dynamic dictionary stores \((key, value)\)-pairs and supports:

\[
\text{add}(key, value), \text{lookup}(key) \quad \text{(which returns value)} \quad \text{and delete}(key)
\]

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by chaining.

A hash function maps a key \(x\) to position \(h(x)\).

\(n\) arbitrary operations arrive online, one at a time.

A set \(H\) of hash functions is weakly universal if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr(h(x) = h(y)) \leq \frac{1}{m}
\]

\((h\ is\ picked\ uniformly\ at\ random\ from\ \(H))\)

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

\(\text{in fact this result can be generalised . . .}\)
A dynamic dictionary stores \((key, value)\)-pairs and supports:

\(\text{add}(key, value), \text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

Universe \(U\) of \(u\) keys.
Hash table \(T\) of size \(m \geq n\).
Collisions are fixed by chaining.

\(n\) arbitrary operations arrive online, one at a time.

A hash function maps
a key \(x\) to position \(h(x)\)

A set \(H\) of hash functions is weakly universal if for any
two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr(h(x) = h(y)) \leq \frac{1}{m}
\]

\((h\) is picked uniformly at random from \(H\))

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

\(\text{in fact this result can be generalised} \ldots\)
A dynamic dictionary stores \((key, value)\)-pairs and supports:

- \(\text{add}(\text{key}, \text{value})\)
- \(\text{lookup}(\text{key})\) (which returns \(\text{value}\))
- \(\text{delete}(\text{key})\)

A hash function maps a key \(x\) to position \(h(x)\).

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by chaining or bucketing.

\(n\) arbitrary operations arrive online, one at a time.

A set \(H\) of hash functions is weakly universal if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr(h(x) = h(y)) \leq \frac{1}{m}
\]

\((h\) is picked uniformly at random from \(H\))

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

\(\text{in fact this result can be generalised . . .}\)
Back to the start (again)

- A **dynamic dictionary** stores *(key, value)*-pairs and supports:
  - `add(key, value)`, `lookup(key)` (which returns `value`) and `delete(key)`

Universe $U$ of $u$ keys.

Hash table $T$ of size $m \geq n$.

Collisions are fixed by **chaining**.

A **hash function** maps a key $x$ to position $h(x)$.

---

$n$ arbitrary operations arrive online, one at a time.

---

A set $H$ of hash functions is **weakly universal** if for any two keys $x, y \in U$ (with $x \neq y$),

$$
\Pr (h(x) = h(y)) \leq \frac{1}{m}
$$

*(h is picked uniformly at random from $H$)*

---

Using weakly universal hashing:

For *any* $n$ operations, the expected run-time is $O(1)$ per operation.

*in fact this result can be generalised*...
A dynamic dictionary stores (key, value)-pairs and supports:

- `add(key, value)`, `lookup(key)` (which returns `value`) and `delete(key)`

**Universe** $U$ of $u$ keys.

**Hash table** $T$ of size $m \geq n$.

Collisions are fixed by **chaining** bucketing.

A hash function maps a key $x$ to position $h(x)$.

$n$ arbitrary operations arrive online, one at a time.

A set $H$ of hash functions is **weakly universal** if for any two keys $x, y \in U$ (with $x \neq y$),

$$\Pr (h(x) = h(y)) \leq \frac{1}{m}$$

($h$ is picked uniformly at random from $H$)

Using weakly universal hashing:

For any $n$ operations, the expected run-time is $O(1)$ per operation.

*in fact this result can be generalised...*
Back to the start (again)

A dynamic dictionary stores \((key, value)\)-pairs and supports:

- \(\text{add}(key, value)\), \(\text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by chaining.

A hash function maps a key \(x\) to position \(h(x)\).

We require that we can recover any key from its bucket in \(O(s)\) time where \(s\) is the number of keys in the bucket.

\(n\) arbitrary operations arrive online, one at a time.

A set \(H\) of hash functions is weakly universal if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr(h(x) = h(y)) \leq \frac{1}{m}
\]

\((h\) is picked uniformly at random from \(H\))

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

\(\text{in fact this result can be generalised . . .}\)
A **dynamic dictionary** stores \((key, value)\)-pairs and supports:

- \(\text{add}(key, value)\), \(\text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by **chaining**.

A hash function maps a key \(x\) to position \(h(x)\).

We require that we can recover any key from its **bucket** in \(O(s)\) time, where \(s\) is the number of keys in the bucket.

\(n\) arbitrary operations arrive online, one at a time.
A dynamic dictionary stores \((key, value)\)-pairs and supports:

\[
\text{add}(key, value), \text{lookup}(key) \text{ (which returns value)} \text{ and delete}(key)
\]

- Universe \(U\) of \(u\) keys.
- Hash table \(T\) of size \(m \geq n\).
- Collisions are fixed by chaining.

We require that we can recover any key from its bucket in \(O(s)\) time where \(s\) is the number of keys in the bucket.

If our construction has the property that, for any two keys \(x, y \in U\) (with \(x \neq y\)), the probability that \(x\) and \(y\) are in the same bucket is at most \(\frac{1}{m}\).
Back to the start (again)

A **dynamic dictionary** stores \((key, value)\)-pairs and supports:

- `add(key, value)`, `lookup(key)` (which returns `value`) and `delete(key)`

Universe \(U\) of \(u\) keys.  
Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by **bucketing**

We require that we can recover any key from its **bucket** in \(O(s)\) time, where \(s\) is the number of keys in the bucket

\(n\) arbitrary operations arrive online, one at a time.

If our construction has the property that, for any two keys \(x, y \in U\) (with \(x \neq y\)), the probability that \(x\) and \(y\) are in the same bucket is at most \(\frac{1}{m}\)

For any \(n\) operations, the *expected* run-time is \(O(1)\) per operation.
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  
  \[
  \text{add}(key, value), \text{lookup}(key) \text{ (which returns value) and delete}(key)
  \]

---

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  \(\text{add}(key, value), \text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

What does *amortised expected \(O(1)\) time* mean?!
Dynamic perfect hashing

A dynamic dictionary stores \((key, value)\)-pairs and supports:

\(\text{add}(key, value), \text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

What does *amortised expected \(O(1)\) time* mean?! *let’s build it up…*
Dynamic perfect hashing

- A **dynamic dictionary** stores *(key, value)*-pairs and supports:
  - `add(key, value)`, `lookup(key)` (which returns `value`) and `delete(key)`

---

**THEOREM**

In the **Cuckoo hashing** scheme:
- Every `lookup` and every `delete` takes $O(1)$ **worst-case** time,
- The space is $O(n)$ where $n$ is the number of keys stored
- An insert takes **amortised expected** $O(1)$ time

What does **amortised expected** $O(1)$ time mean?!

*let’s build it up…*

“$O(1)$ worst-case time per operation”

means every operation takes constant time
Dynamic perfect hashing

- A **dynamic dictionary** stores *(key, value)*-pairs and supports:
  - `add(key, value)`, `lookup(key)` (which returns `value`) and `delete(key)`

**Theorem**

In the **Cuckoo hashing** scheme:

- Every `lookup` and every `delete` takes $O(1)$ **worst-case** time,
- The space is $O(n)$ where $n$ is the number of keys stored
- An `insert` takes **amortised expected** $O(1)$ time

What does *amortised expected $O(1)$ time* mean?!

*let’s build it up…*

"$O(1)$ worst-case time per operation"

means every operation takes constant time

“The total worst-case time complexity of performing any $n$ operations is $O(n)$"
Dynamic perfect hashing

- A **dynamic dictionary** stores \((key, value)\)-pairs and supports:
  - \texttt{add}(key, value)
  - \texttt{lookup}(key) (which returns value)
  - \texttt{delete}(key)

**Theorem**

In the **Cuckoo hashing** scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes \textit{amortised expected} \(O(1)\) time

What does \textit{amortised expected} \(O(1)\) time mean?! **let's build it up...**

- \(O(1)\) worst-case time per operation
  - means every operation takes constant time

- “The total worst-case time complexity of performing any \(n\) operations is \(O(n)\)”
  - this **does not** imply that every operation takes constant time
Dynamic perfect hashing

A dynamic dictionary stores \((key, value)\)-pairs and supports:

- \(\text{add}(key, value)\), \(\text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

**THEOREM**

In the Cuckoo hashing scheme:

- Every \(\text{lookup}\) and every \(\text{delete}\) takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An \(\text{insert}\) takes amortised expected \(O(1)\) time

What does *amortised expected \(O(1)\) time* mean?! *let’s build it up…*

“\(O(1)\) worst-case time per operation”

means every operation takes constant time

“The total worst-case time complexity of performing any \(n\) operations is \(O(n)\)”

this does not imply that every operation takes constant time

However, it does mean that the *amortised worst-case* time complexity of an operation is \(O(1)\)
Dynamic perfect hashing

A dynamic dictionary stores \((key, value)\)-pairs and supports:

\[\text{add}(key, value), \text{lookup}(key)\ (\text{which returns } value) \text{ and } \text{delete}(key)\]

---

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

What does *amortised expected \(O(1)\)* time mean?! *let’s build it up...*

*\(O(1)\) expected time per operation* means every operation takes constant time in expectation

*The total expected time complexity of performing any \(n\) operations is \(O(n)\)*

this does not imply that every operation takes constant time in expectation

However, it does mean that the amortised expected time complexity of an operation is \(O(1)\)*
A dynamic dictionary stores \((key, value)\)-pairs and supports:

- \(\text{add}(key, value)\), \(\text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

**Theorem**

In the Cuckoo hashing scheme:

- Every \(\text{lookup}\) and every \(\text{delete}\) takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  
  \[
  \text{add}(key, value), \text{lookup}(key) \text{ (which returns value)} \text{ and delete}(key)
  \]

---

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  
  - `add(key, value)`, `lookup(key)` (which returns `value`) and `delete(key)`

---

**THEOREM**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) **worst-case** time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes **amortised expected** \(O(1)\) time

In Cuckoo hashing there is a single hash table but **two** hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).
Dynamic perfect hashing

- **A dynamic dictionary** stores \((key, value)\)-pairs and supports:
  
  \[
  \text{add}(key, value), \text{lookup}(key) \quad \text{(which returns value)} \quad \text{and delete}(key)
  \]

**Theorem**

In the **Cuckoo hashing** scheme:

- Every **lookup** and every **delete** takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An **insert** takes amortised expected \(O(1)\) time

In Cuckoo hashing there is a single hash table but **two** hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  
  \(\text{add}(\text{key, value}), \text{lookup}(\text{key})\) (which returns \text{value}) and \text{delete}(\text{key})

---

**Theorem**

In the Cuckoo hashing scheme:
- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).
Dynamic perfect hashing

A dynamic dictionary stores \((key, value)\)-pairs and supports:

\[
\text{add}(key, value), \text{lookup}(key) \text{ (which returns value) and delete}(key)
\]

**THEOREM**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).
Dynamic perfect hashing

A dynamic dictionary stores \((key, value)\)-pairs and supports:

\quad \text{add}(key, value), \text{lookup}(key) \quad \text{(which returns value)} \quad \text{and delete}(key)

\textbf{THEOREM}

In the \textbf{Cuckoo hashing} scheme:

- Every \textit{lookup} and every \textit{delete} takes \(O(1)\) \textit{worst-case} time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes \textit{amortised expected} \(O(1)\) time

In \textbf{Cuckoo hashing} there is a single hash table but \textbf{two} hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  
  \[
  \text{add}(key, value), \text{lookup}(key) \text{ (which returns } value) \text{ and delete}(key)
  \]

---

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes *amortised expected* \(O(1)\) time

---

In Cuckoo hashing there is a single hash table but **two** hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).

**Important**: We never store multiple keys at the same position
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  
  \[
  \text{add}(key, value), \text{lookup}(key) \text{ (which returns value)} \text{ and delete}(key)
  \]

---

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

---

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).

**Important:** We never store multiple keys at the same position

\[
\begin{align*}
&h_1(x) & &h_2(x) \\
&\text{Therefore, as claimed, lookup takes } O(1) \text{ time} \ldots
\end{align*}
\]
Dynamic perfect hashing

A dynamic dictionary stores \((key, value)\)-pairs and supports:

- add\((key, value)\), lookup\((key)\) (which returns \(value\)) and delete\((key)\)

**Theorem**

In the Cuckoo hashing scheme:
- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).

**Important**: We never store multiple keys at the same position

\[h_1(x)\]

\[h_2(x)\]

Therefore, as claimed, lookup takes \(O(1)\) time... but how do we do inserts?
Inserts in Cuckoo hashing

\[ h_1(x) \]

\[ h_2(x) \]

Step 1: Attempt to put \( x \) in position \( h_1(x) \)
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$

if that position is empty, stop (and congratulate yourself on a job well done)
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$

if that position is empty, stop
Inserts in Cuckoo hashing

**Step 1**: Attempt to put $x$ in position $h_1(x)$

*if that position is empty, stop*
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$

if that position is empty, stop
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$

if that position is empty, stop

Step 2: Let $y$ be the key currently in position $h_1(x)$
Inserts in Cuckoo hashing

Step 1: Attempt to put \( x \) in position \( h_1(x) \)
if that position is empty, stop

Step 2: Let \( y \) be the key currently in position \( h_1(x) \)
evict key \( y \) and replace it with key \( x \)
**Inserts in Cuckoo hashing**

**Step 1**: Attempt to put $x$ in position $h_1(x)$

*if that position is empty, stop*

**Step 2**: Let $y$ be the key currently in position $h_1(x)$

evict key $y$ and replace it with key $x$
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$
   *if that position is empty, stop*

Step 2: Let $y$ be the key currently in position $h_1(x)$
   evict key $y$ and replace it with key $x$

*where should we put key $y$?*
Inserts in Cuckoo hashing

**Step 1:** Attempt to put $x$ in position $h_1(x)$

*if that position is empty, stop*

**Step 2:** Let $y$ be the key currently in position $h_1(x)$

evict key $y$ and replace it with key $x$

*where should we put key $y$?*

in the *other* position it’s allowed in
Inserts in Cuckoo hashing

**Step 1:** Attempt to put \( x \) in position \( h_1(x) \)

*if that position is empty, stop*

**Step 2:** Let \( y \) be the key currently in position \( h_1(x) \)

evict key \( y \) and replace it with key \( x \)

*where should we put key \( y \)?*

in the *other* position it’s allowed in
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$
   if that position is empty, stop

Step 2: Let $y$ be the key currently in position $h_1(x)$
   evict key $y$ and replace it with key $x$

Step 3: Let $pos$ be the other position $y$ is allowed to be in
   i.e $pos = h_2(y)$ if $h_1(x) = h_1(y)$ and $pos = h_1(y)$ otherwise
Inserts in Cuckoo hashing

**Step 1:** Attempt to put $x$ in position $h_1(x)$

*if that position is empty, stop*

**Step 2:** Let $y$ be the key currently in position $h_1(x)$

*evict key $y$ and replace it with key $x$*

**Step 3:** Let $pos$ be the other position $y$ is allowed to be in

*i.e $pos = h_2(y)$ if $h_1(x) = h_1(y)$ and $pos = h_1(y)$ otherwise*

**Step 4:** Attempt to put $y$ in position $pos$

*if that position is empty, stop*
 Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$
if that position is empty, stop

Step 2: Let $y$ be the key currently in position $h_1(x)$
evict key $y$ and replace it with key $x$

Step 3: Let $pos$ be the other position $y$ is allowed to be in
i.e $pos = h_2(y)$ if $h_1(x) = h_1(y)$ and $pos = h_1(y)$ otherwise

Step 4: Attempt to put $y$ in position $pos$
if that position is empty, stop
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$
if that position is empty, stop

Step 2: Let $y$ be the key currently in position $h_1(x)$
evict key $y$ and replace it with key $x$

Step 3: Let $pos$ be the other position $y$ is allowed to be in
i.e $pos = h_2(y)$ if $h_1(x) = h_1(y)$ and $pos = h_1(y)$ otherwise

Step 4: Attempt to put $y$ in position $pos$
if that position is empty, stop
Inserts in Cuckoo hashing

**Step 1:** Attempt to put \( x \) in position \( h_1(x) \)

*if that position is empty, stop*

**Step 2:** Let \( y \) be the key currently in position \( h_1(x) \)

evict key \( y \) and replace it with key \( x \)

**Step 3:** Let \( \text{pos} \) be the other position \( y \) is allowed to be in

\( \text{i.e } \text{pos} = h_2(y) \) if \( h_1(x) = h_1(y) \) and \( \text{pos} = h_1(y) \) otherwise

**Step 4:** Attempt to put \( y \) in position \( \text{pos} \)

*if that position is empty, stop*

**Step 5:** Let \( z \) be the key currently in position \( \text{pos} \)

evict key \( z \) and replace it with key \( y \)
**Inserts in Cuckoo hashing**

Step 1: Attempt to put $x$ in position $h_1(x)$

*if that position is empty, stop*

Step 2: Let $y$ be the key currently in position $h_1(x)$

evict key $y$ and replace it with key $x$

Step 3: Let $pos$ be the *other* position $y$ is allowed to be in

*i.e.* $pos = h_2(y)$ if $h_1(x) = h_1(y)$ and $pos = h_1(y)$ otherwise

Step 4: Attempt to put $y$ in position $pos$

*if that position is empty, stop*

Step 5: Let $z$ be the key currently in position $pos$

evict key $z$ and replace it with key $y$
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$
if that position is empty, stop

Step 2: Let $y$ be the key currently in position $h_1(x)$
evict key $y$ and replace it with key $x$

Step 3: Let $pos$ be the other position $y$ is allowed to be in
i.e $pos = h_2(y)$ if $h_1(x) = h_1(y)$ and $pos = h_1(y)$ otherwise

Step 4: Attempt to put $y$ in position $pos$
if that position is empty, stop

Step 5: Let $z$ be the key currently in position $pos$
evict key $z$ and replace it with key $y$
Inserts in Cuckoo hashing

**Step 1:** Attempt to put \( x \) in position \( h_1(x) \)

*if that position is empty, stop*

**Step 2:** Let \( y \) be the key currently in position \( h_1(x) \)

evict key \( y \) and replace it with key \( x \)

**Step 3:** Let \( \text{pos} \) be the *other* position \( y \) is allowed to be in

\[ i.e \ pos = h_2(y) \text{ if } h_1(x) = h_1(y) \text{ and } pos = h_1(y) \text{ otherwise} \]

**Step 4:** Attempt to put \( y \) in position \( \text{pos} \)

*if that position is empty, stop*

**Step 5:** Let \( z \) be the key currently in position \( \text{pos} \)

evict key \( z \) and replace it with key \( y \)  
*and so on...*
Pseudocode

\textbf{add}(x):

\begin{itemize}
  \item \(\text{pos} \leftarrow h_1(x)\)
  \item repeat at most \(n\) times:
    \begin{itemize}
      \item If \(T[\text{pos}]\) is empty then \(T[\text{pos}] \leftarrow x\).
      \item Otherwise, \(y \leftarrow T[\text{pos}]\), \(T[\text{pos}] \leftarrow x\), \(\text{pos} \leftarrow\) the other possible location for \(y\).
    \end{itemize}
  \item \(x \leftarrow y\).
  \end{itemize}

Repeat

\begin{itemize}
  \item Give up and rehash the whole table.
    \textit{i.e. empty the table, pick two new hash functions and reinsert every key}
\end{itemize}
Rehashing

If we fail to insert a new key $x$,

(i.e. we still have an “evicted” key after moving around keys $n$ times)

then we declare the table “rubbish” and rehash.
Rehashing

If we fail to insert a new key $x$,

(i.e. we still have an “evicted” key after moving around keys $n$ times)
then we declare the table “rubbish” and rehash.

What does rehashing involve?
Rehashing

If we fail to insert a new key $x$,

(i.e. we still have an “evicted” key after moving around keys $n$ times)

then we declare the table “rubbish” and rehash.

What does rehashing involve?

Suppose that the table contains the $k$ keys $x_1, \ldots, x_k$

at the time of we fail to insert key $x$. 
Rehashing

If we fail to insert a new key $x$,

(i.e. we still have an “evicted” key after moving around keys $n$ times)

then we declare the table “rubbish” and rehash.

What does rehashing involve?

Suppose that the table contains the $k$ keys $x_1, \ldots, x_k$

at the time of we fail to insert key $x$.

To rehash we:
Rehashing

If we fail to insert a new key \( x \),

\( i.e. \text{ we still have an “evicted” key after moving around keys } n \text{ times} \)

then we declare the table “rubbish” and rehash.

What does rehashing involve?

Suppose that the table contains the \( k \) keys \( x_1, \ldots, x_k \)

at the time of we fail to insert key \( x \).

To rehash we:

Randomly pick two new hash functions \( h_1 \) and \( h_2 \). (More about this in a minute.)
Rehashing

If we fail to insert a new key \( x \),

(i.e. we still have an “evicted” key after moving around keys \( n \) times)
then we declare the table “rubbish” and rehash.

What does rehashing involve?

Suppose that the table contains the \( k \) keys \( x_1, \ldots, x_k \)
at the time of we fail to insert key \( x \).

To rehash we:

Randomly pick two new hash functions \( h_1 \) and \( h_2 \). (More about this in a minute.)

Build a new, empty, hash table of the same size
Rehashing

If we fail to insert a new key $x$,  
(i.e. we still have an “evicted” key after moving around keys $n$ times)  
then we declare the table “rubbish” and rehash.

What does rehashing involve?

Suppose that the table contains the $k$ keys $x_1, \ldots, x_k$  
at the time of we fail to insert key $x$.

To rehash we:

Randomly pick two new hash functions $h_1$ and $h_2$. (More about this in a minute.)

Build a *new*, empty, hash table of the same size

*Reinsert* the keys $x_1, \ldots, x_k$ and then $x$,  
one by one, using the normal add operation.
Rehashing

If we fail to insert a new key \( x \),

\[ \text{(i.e. we still have an “evicted” key after moving around keys n times)} \]

then we declare the table “rubbish” and rehash.

**What does rehashing involve?**

Suppose that the table contains the \( k \) keys \( x_1, \ldots, x_k \) at the time of we fail to insert key \( x \).

To rehash we:

Randomly pick two new hash functions \( h_1 \) and \( h_2 \). (More about this in a minute.)

Build a *new*, empty, hash table of the same size

*Reinsert* the keys \( x_1, \ldots, x_k \) and then \( x \),

*one by one, using the normal add operation.*

If we fail while rehashing... we start from the beginning again
Rehashing

If we fail to insert a new key \( x \),

\( i.e. \) we still have an “evicted” key after moving around keys \( n \) times

then we declare the table “rubbish” and rehash.

What does rehashing involve?

Suppose that the table contains the \( k \) keys \( x_1, \ldots, x_k \)

at the time of we fail to insert key \( x \).

To rehash we:

Randomly pick two new hash functions \( h_1 \) and \( h_2 \). (More about this in a minute.)

Build a new, empty, hash table of the same size

Reinsert the keys \( x_1, \ldots, x_k \) and then \( x \),

one by one, using the normal add operation.

If we fail while rehashing... we start from the beginning again

This is rather slow... but we will prove that it happens rarely
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh *(see the link on unit web page).*
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh *(see the link on unit web page)*.

We make the following assumptions:
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh (*see the link on unit web page*).

We make the following assumptions:

\[ h_1 \text{ and } h_2 \text{ are independent} \]
\[ \text{i.e. } h_1(x) \text{ says nothing about } h_2(x), \text{ and vice versa.} \]
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh (see the link on unit web page).

We make the following assumptions:

\( h_1 \) and \( h_2 \) are independent
\hspace{1cm} \text{i.e.} \( h_1(x) \) says nothing about \( h_2(x) \), and vice versa.

\( h_1 \) and \( h_2 \) are truly random
\hspace{1cm} \text{i.e.} each key is independently mapped to a particular position
\hspace{1cm} \text{in the hash table with probability} \ \frac{1}{m}.
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh (see the link on unit web page).

We make the following assumptions:

1. **$h_1$ and $h_2$ are truly random**
   - i.e. $h_1(x)$ says nothing about $h_2(x)$, and vice versa.

2. **$h_1$ and $h_2$ are independent**
   - i.e. each key is independently mapped to a particular position in the hash table with probability $\frac{1}{m}$.

Computing the value of $h_1(x)$ and $h_2(x)$ takes $O(1)$ worst-case time.
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh (see the link on unit web page).

We make the following assumptions:

$h_1$ and $h_2$ are truly random
i.e. $h_1(x)$ says nothing about $h_2(x)$, and vice versa.

$h_1$ and $h_2$ are independent
i.e. each key is independently mapped to a particular position in the hash table with probability $\frac{1}{m}$.

Computing the value of $h_1(x)$ and $h_2(x)$ takes $O(1)$ worst-case time.

There are at most $n$ keys in the hash table at any time.
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh *(see the link on unit web page)*.

We make the following assumptions:

\( h_1 \) and \( h_2 \) are truly random
i.e. \( h_1(x) \) says nothing about \( h_2(x) \), and vice versa.

\( h_1 \) and \( h_2 \) are independent
i.e. each key is independently mapped to a particular position in the hash table with probability \( \frac{1}{m} \).

Computing the value of \( h_1(x) \) and \( h_2(x) \) takes \( O(1) \) worst-case time

*There are at most \( n \) keys in the hash table at any time.*
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh (see the link on unit web page).

We make the following assumptions:

- $h_1$ and $h_2$ are independent
  - i.e. $h_1(x)$ says nothing about $h_2(x)$, and vice versa.

- $h_1$ and $h_2$ are truly random
  - i.e. each key is independently mapped to a particular position in the hash table with probability $\frac{1}{m}$.

Computing the value of $h_1(x)$ and $h_2(x)$ takes $O(1)$ worst-case time

There are at most $n$ keys in the hash table at any time.
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh (see the link on unit web page).

We make the following assumptions:

- **Reasonable Assumption**
  \[ h_1 \text{ and } h_2 \text{ are independent} \]
  i.e. \( h_1(x) \) says nothing about \( h_2(x) \), and vice versa.

- **Unreasonable Assumption**
  \[ h_1 \text{ and } h_2 \text{ are truly random} \]
  i.e. each key is independently mapped to a particular position in the hash table with probability \( \frac{1}{m} \).

- **Questionable Assumption**
  Computing the value of \( h_1(x) \) and \( h_2(x) \) takes \( O(1) \) worst-case time.

*There are at most \( n \) keys in the hash table at any time.*
We make the following assumptions:

- **Reasonable Assumption**: $h_1$ and $h_2$ are independent
  i.e. $h_1(x)$ says nothing about $h_2(x)$, and vice versa.

- **Unreasonable Assumption**: $h_1$ and $h_2$ are truly random
  i.e. each key is independently mapped to a particular position
  in the hash table with probability $\frac{1}{m}$.

- **Questionable Assumption**: Computing the value of $h_1(x)$ and $h_2(x)$ takes $O(1)$ worst-case time

There are at most $n$ keys in the hash table at any time.
Cuckoo graph

Hash table
(size $m$)
Cuckoo graph

Hash table
(size $m$)

The **cuckoo graph**: 
Cuckoo graph

Hash table
(size $m$)

The **cuckoo graph**:

A vertex for each position of the table.
Cuckoo graph

The **cuckoo graph**: 
A vertex for each position of the table.
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table. For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

**Hash table**
(size $m$)

- $x_1$
- $h_1(x_1)$
- $h_2(x_1)$

$m$ vertices
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 
The **cuckoo graph**: 

A vertex for each position of the table. 

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

---

**Cuckoo graph**

Hash table (size $m$)

$m$ vertices
The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.
The **cuckoo graph**: A vertex for each position of the table. For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

Cuckoo graph

**Hash table** (size $m$)

$m$ vertices

$x_4$  
$x_3$  
$x_2$  

$x_1$  
$h_1(x_1)$  

$h_2(x_1)$  

$x_5$
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

---

**Cuckoo graph**

Hash table (size $m$)

$m$ vertices

$x_4$

$x_3$

$x_2$

$x_1$

$x_5$
Cuckoo graph

The **cuckoo graph**: A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 
The **cuckoo graph**:

A vertex for each position of the table.

For each key \( x \) there is an undirected edge between \( h_1(x) \) and \( h_2(x) \).

*The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph*
The **cuckoo graph**:  
A vertex for each position of the table. 
For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

*The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph*
The **cuckoo graph**:  
A vertex for each position of the table. 
For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph

Inserting key $x_6$ causes a cycle.
The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

*The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph*

Inserting key $x_6$ causes a cycle.

*Cycles are dangerous…*
The **cuckoo graph**: A vertex for each position of the table. For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph. Inserting key $x_6$ causes a cycle. Cycles are dangerous…
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.

Inserting key $x_6$ causes a cycle. *Cycles are dangerous*…

Inserting the key $x_7$ triggers a rehash,
The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.

Inserting key $x_6$ causes a cycle. *Cycles are dangerous…*

Inserting the key $x_7$ triggers a rehash, because the keys will be moved around in an infinite loop (but we stop and rehash after $n$ moves).
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.

Inserting key $x_6$ causes a cycle.

Cycles are dangerous…

Inserting the key $x_7$ triggers a rehash, because the keys will be moved around in an infinite loop (but we stop and rehash after $n$ moves).

**here there are 6 keys but only 5 spaces**
The **cuckoo graph**: A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

*The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.*

Inserting key $x_6$ causes a cycle.

*Cycles are dangerous…*

Inserting the key $x_7$ triggers a rehash, because the keys will be moved around in an infinite loop (but we stop and rehash after $n$ moves).

*Here there are 6 keys but only 5 spaces*

We will analyse the probability of a cycle or a long path occurring while inserting any $n$ keys.
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.
**Paths in the cuckoo graph**

**Lemma**
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

**Proof**
Proof by induction.

**Base case:** $\ell = 1$. 

![Diagram of cuckoo graph with key $x$ and positions $i$ and $j$.]
Paths in the cuckoo graph

**Lemma**

For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

**Proof**

Proof by induction.

**Base case:** $\ell = 1$.

Let $K$ be the set of keys in the hash table. $|K| \leq n$. 
Paths in the cuckoo graph

**Lemma**

For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c\ell \cdot m}$.

**Proof**

Proof by induction.

**Base case:** $\ell = 1$.

Let $K$ be the set of keys in the hash table. $|K| \leq n$.

The probability that a key $x$ is mapped to positions $i$ and $j$,

i.e. either $h_1(x) = i$, $h_2(x) = j$ or $h_1(x) = j$, $h_2(x) = i$,

is at most $\frac{2}{m^2}$ (*recall we have assumed independence between $h_1$ and $h_2$)*
Paths in the cuckoo graph

**Lemma**
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

**Proof**
Proof by induction.

**Base case:** $\ell = 1$.

Let $K$ be the set of keys in the hash table. $|K| \leq n$.

The probability that a key $x$ is mapped to positions $i$ and $j$,

i.e. either $h_1(x) = i$, $h_2(x) = j$ or $h_1(x) = j$, $h_2(x) = i$,

is at most $\frac{2}{m^2}$ *(recall we have assumed independence between $h_1$ and $h_2$)*

Therefore *(using the union bound)* the probability that there is an edge between $i$ and $j$ is at most

$$\sum_{x \in K} \frac{2}{m^2} \leq \frac{2n}{m^2} \leq \frac{1}{c \cdot m}.$$ 

since $m \geq 2cn$. 
Inductive step: assume lemma is true for lengths $1, 2, \ldots, \ell - 1$. 

Proof continued...
Paths in the cuckoo graph

**Proof continued...**

**Inductive step:** assume lemma is true for lengths $1, 2, \ldots, \ell - 1$.

If there is a path between $i$ and $j$ of length $\ell$ but *not shorter* than $\ell$
then there must be a position $k$ such that:
**Proof continued...**

**Inductive step:** assume lemma is true for lengths $1, 2, \ldots, \ell - 1$.

If there is a path between $i$ and $j$ of length $\ell$ but *not shorter* than $\ell$
then there must be a position $k$ such that:

A  there is a shortest path of length $\ell - 1$ from $i$ to $k$ that does not go through $j$, and

B  there is an edge from $k$ to $j$. 
Inductive step: assume lemma is true for lengths 1, 2, . . . , \(\ell - 1\).

If there is a path between \(i\) and \(j\) of length \(\ell\) but *not shorter* than \(\ell\)
then there must be a position \(k\) such that:

A  there is a shortest path of length \(\ell - 1\) from \(i\) to \(k\) that does not go through \(j\),

and

B  there is an edge from \(k\) to \(j\).

By the inductive hypothesis,

\[
\Pr(A) \leq \frac{1}{c^{\ell-1}m}.
\]
**Inductive step:** assume lemma is true for lengths 1, 2, \ldots, \ell - 1.

If there is a path between \(i\) and \(j\) of length \(\ell\) but *not shorter* than \(\ell\)
then there must be a position \(k\) such that:

- **A** there is a shortest path of length \(\ell - 1\) from \(i\) to \(k\) that does not go through \(j\),

  and

- **B** there is an edge from \(k\) to \(j\).

By the inductive hypothesis,
\[
\Pr(A) \leq \frac{1}{c^{\ell-1} \cdot m}.
\]

**Observe** The “not go through \(j\)” can only make the probability smaller.

Given that **A** is true, the probability that **B** holds as well is at most
\[
\sum_{x \in K} \frac{2}{m^2} \leq \frac{1}{c \cdot m}. \quad \text{(union bound like on the previous slide over keys in } K)\]
**Paths in the cuckoo graph**

**Proof continued...**

**Inductive step:** assume lemma is true for lengths $1, 2, \ldots, \ell - 1$.

If there is a path between $i$ and $j$ of length $\ell$ but *not shorter* than $\ell$ then there must be a position $k$ such that:

- **A** there is a shortest path of length $\ell - 1$ from $i$ to $k$ that does not go through $j$,
- and
- **B** there is an edge from $k$ to $j$.

By the inductive hypothesis,

$$\Pr(A) \leq \frac{1}{c^{\ell-1} \cdot m}.$$  

**Observe** The "not go through $j$" can only make the probability smaller.

Given that $A$ is true, the probability that $B$ holds as well is at most

$$\sum_{x \in K} \frac{2}{m^2} \leq \frac{1}{c \cdot m}. \quad \text{(union bound like on the previous slide over keys in $K$.)}$$

$$\Pr(A \text{ and } B) = \Pr(A) \cdot \Pr(B \mid A) \leq \frac{1}{c^{\ell-1} \cdot m} \cdot \frac{1}{c \cdot m} = \frac{1}{c^\ell \cdot m^2}.$$
**Paths in the cuckoo graph**

**Proof continued...**

**Inductive step:** assume lemma is true for lengths 1, 2, \ldots, \(\ell - 1\).

If there is a path between \(i\) and \(j\) of length \(\ell\) but *not shorter* than \(\ell\) then there must be a position \(k\) such that:

**A**  there is a shortest path of length \(\ell - 1\) from \(i\) to \(k\) that does not go through \(j\),

and

**B**  there is an edge from \(k\) to \(j\).

By the inductive hypothesis,

\[
\Pr(A) \leq \frac{1}{c^{\ell-1} \cdot m}.
\]

**Observe** The "not go through \(j\)" can only make the probability smaller.

Given that \(A\) is true, the probability that \(B\) holds as well is at most

\[
\sum_{x \in K} \frac{2}{m^2} \leq \frac{1}{c \cdot m}. \quad \text{(union bound like on the previous slide over keys in } K)\]

\[
\Pr(A\text{ and }B) = \Pr(A) \cdot \Pr(B \mid A) \leq \frac{1}{c^{\ell-1} \cdot m} \cdot \frac{1}{c \cdot m} = \frac{1}{c^{\ell} \cdot m^2}.
\]

The union bound over all ‘midpoints’ \(k\) gives an upper bound on the probability of a shortest path between \(i\) and \(j\) of length \(\ell\):

\[
\leq m \cdot \frac{1}{c^{\ell} \cdot m^2} = \frac{1}{c^{\ell} \cdot m}.
\]
Back to buckets

We say that two keys $x, y$ are in the same **bucket** (conceptually) iff there is a path between $h_1(x)$ and $h_1(y)$ in the cuckoo graph.
Back to buckets

We say that two keys $x, y$ are in the same bucket (conceptually) iff there is a path between $h_1(x)$ and $h_1(y)$ in the cuckoo graph.

For two distinct keys $x, y$, the probability that they are in the same bucket is at most

$$4 \sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{4}{m(c - 1)} = O\left(\frac{1}{m}\right)$$

where $c > 1$ is a constant.

(Another union bound over all possible path lengths.)
Back to buckets

We say that two keys $x, y$ are in the same bucket (conceptually)
iff there is a path between $h_1(x)$ and $h_1(y)$
in the cuckoo graph.

For two distinct keys $x, y$, the probability
that they are in the same bucket is at most

$$4 \sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{4}{m(c-1)} = O\left(\frac{1}{m}\right)$$

where $c > 1$ is a constant.

(another union bound over all possible path lengths.)

The time for an operation on $x$ is bounded by
the number of items in the bucket. (Assuming there are no cycles.)
Back to buckets

We say that two keys $x, y$ are in the same **bucket** (conceptually) iff there is a path between $h_1(x)$ and $h_1(y)$ in the cuckoo graph.

For two distinct keys $x, y$, the probability that they are in the same bucket is at most

$$4 \sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{4}{m(c - 1)} = O\left(\frac{1}{m}\right)$$

where $c > 1$ is a constant.

*(another union bound over all possible path lengths.)*

The time for an operation on $x$ is bounded by the number of items in the bucket. *(Assuming there are no cycles.)*

So we have that the expected time per operation is $O(1)$ *(assuming that $m \geq 2cn$).*
Back to buckets

We say that two keys $x, y$ are in the same **bucket** (conceptually) iff there is a path between $h_1(x)$ and $h_1(y)$ in the cuckoo graph.

For two distinct keys $x, y$, the probability that they are in the same bucket is at most

$$4 \sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{4}{m(c-1)} = O\left(\frac{1}{m}\right)$$

where $c > 1$ is a constant.

(another union bound over all possible path lengths.)

The time for an operation on $x$ is bounded by the number of items in the bucket. *(Assuming there are no cycles.)*

So we have that the expected time per operation is $O(1)$ (assuming that $m \geq 2cn$).

Further, lookups take $O(1)$ time in the **worst case**.
Rehashing

The previous analysis on the expected running time holds when there are *no cycles*.
Rehashing

The previous analysis on the expected running time holds when there are no cycles.

However, we would expect there to be cycles every now and then, causing a rehash.
Rehashing

The previous analysis on the expected running time holds when there are *no cycles*.

However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen?
Rehashing

The previous analysis on the expected running time holds when there are no cycles.

However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen?

For simplicity, let us assume that there are already \( n \) keys in the table and we want to insert another \( n \) keys.

We assume now that the table size \( m \geq 4cn \), where \( c > 1 \) is the constant from the previous slides.
Rehashing

The previous analysis on the expected running time holds when there are \textit{no cycles}.

However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen?

For simplicity, let us assume that there are already \( n \) keys in the table and we want to insert \textit{another} \( n \) keys.

We assume now that the table size \( m \geq 4cn \), where \( c > 1 \) is the constant from the previous slides.

A cycle is a path from a vertex \( i \) back to itself.
Rehashing

The previous analysis on the expected running time holds when there are \textit{no cycles}.

However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen?

For simplicity, let us assume that there are already \( n \) keys in the table and we want to insert \textit{another} \( n \) keys.

We assume now that the table size \( m \geq 4cn \),

where \( c > 1 \) is the constant from the previous slides.

A cycle is a path from a vertex \( i \) back to itself.

so use previous result with \( i = j \) ….
Rehashing

The previous analysis on the expected running time holds when there are \textit{no cycles}.

However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen?

For simplicity, let us assume that there are already \(n\) keys in the table and we want to insert \textit{another} \(n\) keys.

We assume now that the table size \(m \geq 4cn\), where \(c > 1\) is the constant from the previous slides.

A cycle is a path from a vertex \(i\) back to itself.

so use previous result with \(i = j\)...

**Lemma**

For any positions \(i\) and \(j\), and any constant \(c > 1\), if \(m \geq 2cn\) then the probability that there exists a shortest path in the cuckoo graph from \(i\) to \(j\) with length \(\ell \geq 1\), is at most \(\frac{1}{c^\ell \cdot m}\).
Rehashing

The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c - 1)}.$$  

(another union bound over all possible path lengths.)
The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{c^\ell \cdot m}{m(c-1)} = \frac{1}{m(c-1)}.$$ (another union bound over all possible path lengths.)

The probability that there is at least one cycle is at most

$$m \cdot \frac{1}{m(c-1)} = \frac{1}{c-1}.$$
The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c - 1)}.$$  

(Another union bound over all possible path lengths.)

The probability that there is at least one cycle is at most

$$m \cdot \frac{1}{m(c - 1)} = \frac{1}{c - 1}.$$  

(Union bound over all $m$ positions in the table.)
Rehashing

The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c-1)}.$$  

(union bound over all possible path lengths.)

The probability that there is at least one cycle is at most

$$m \cdot \frac{1}{m(c-1)} = \frac{1}{c-1}.$$  

(union bound over all $m$ positions in the table.)

If we set $c = 3$, the probability is at most $\frac{1}{2}$ that a cycle occurs

(that there is a rehash) during the $n$ insertions.
Rehashing

The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c - 1)}.$$  

(Another union bound over all possible path lengths.)

The probability that there is at least one cycle is at most

$$m \cdot \frac{1}{m(c - 1)} = \frac{1}{c - 1}.$$  

(Union bound over all $m$ positions in the table.)

If we set $c = 3$, the probability is at most $\frac{1}{2}$ that a cycle occurs  

(that there is a rehash) during the $n$ insertions.

The probability that there are two rehashes is therefore $\frac{1}{4}$, and so on.
Rehashing

The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c-1)}.$$  

(another union bound over all possible path lengths.)

The probability that there is at least one cycle is at most

$$m \cdot \frac{1}{m(c-1)} = \frac{1}{c-1}.$$  

(union bound over all $m$ positions in the table.)

If we set $c = 3$, the probability is at most $\frac{1}{2}$ that a cycle occurs

(that there is a rehash) during the $n$ insertions.

The probability that there are two rehashes is therefore $\frac{1}{4}$, and so on.

So the expected number of rehashes during $n$ insertions is at most $\sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^i = 1$. 
Rehashing

If the expected time for one rehash is $O(n)$ then
the expected time for all rehashes is also $O(n)$

*(this is because we only expect there to be one rehash).*
Rehashing

If the expected time for one rehash is $O(n)$ then

the expected time for all rehashes is also $O(n)$

(this is because we only expect there to be one rehash).

Therefore the *amortised expected* time for the rehashes over the $n$ insertions is $O(1)$ per insertion (i.e. divide the total cost with $n$).
Rehashing

If the expected time for one rehash is $O(n)$ then
the expected time for all rehashes is also $O(n)$
(this is because we only expect there to be one rehash).

Therefore the *amortised expected* time for the rehashes over the $n$ insertions is
$O(1)$ per insertion (i.e. divide the total cost with $n$).

Why is the expected time per rehash $O(n)$?
Rehashing

If the expected time for one rehash is $O(n)$ then

the expected time for all rehashes is also $O(n)$

(this is because we only expect there to be one rehash).

Therefore the *amortised expected* time for the rehashes over the $n$ insertions is

$O(1)$ per insertion (i.e. divide the total cost with $n$).

Why is the expected time per rehash $O(n)$?

First pick a new random $h_1$ and $h_2$ and construct the cuckoo graph using the at most $n$ keys.
Rehashing

If the expected time for one rehash is $O(n)$ then
the expected time for all rehashes is also $O(n)$
\((this \ is \ because \ we \ only \ expect \ there \ to \ be \ one \ rehash)\).

Therefore the *amortised expected* time for the rehashes over the $n$ insertions is
$O(1)$ per insertion (i.e. divide the total cost with $n$).

Why is the expected time per rehash $O(n)$?

First pick a new random $h_1$ and $h_2$ and construct the cuckoo graph
using the at most $n$ keys.

Check for a cycle in the graph in $O(n)$ time (and start again if you find one)
Rehashing

If the expected time for one rehash is $O(n)$ then
the expected time for all rehashes is also $O(n)$
(this is because we only expect there to be one rehash).

Therefore the *amortised expected* time for the rehashes over the $n$ insertions is
$O(1)$ per insertion (i.e. divide the total cost with $n$).

Why is the expected time per rehash $O(n)$?

First pick a new random $h_1$ and $h_2$ and construct the cuckoo graph
using the at most $n$ keys.

Check for a cycle in the graph in $O(n)$ time (and start again if you find one)
(you can do this using breadth-first search)
Rehashing

If the expected time for one rehash is $O(n)$ then
the expected time for all rehashes is also $O(n)$
\[(this\ is\ because\ we\ only\ expect\ there\ to\ be\ one\ rehash).\]

Therefore the *amortised expected* time for the rehashes over the $n$ insertions is $O(1)$ per insertion (i.e. divide the total cost with $n$).

Why is the expected time per rehash $O(n)$?

First pick a new random $h_1$ and $h_2$ and construct the cuckoo graph
using the at most $n$ keys.

Check for a cycle in the graph in $O(n)$ time (and start again if you find one)
\[(you\ can\ do\ this\ using\ breadth-first\ search)\]

If there is no cycle, insert all the elements,
this takes $O(n)$ time in expectation \[(as\ we\ have\ seen).\]
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

**THEOREM**

In the **Cuckoo hashing** scheme:

- Every **lookup** and every **delete** takes $O(1)$ **worst-case** time,
- The space is $O(n)$ where $n$ is the number of keys stored
- An **insert** takes **amortised expected** $O(1)$ time
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic

where any two keys \( x, y \) are independent

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \( O(1) \) worst-case time,
- The space is \( O(n) \) where \( n \) is the number of keys stored
- An insert takes amortised expected \( O(1) \) time
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic where any two keys $x, y$ are independent.

We can define a stronger hash families with $k$-independence.

here the hash values of any choice of $k$ keys are independent.

---

**THEOREM**

In the **Cuckoo hashing** scheme:

- Every lookup and every delete takes $O(1)$ worst-case time,
- The space is $O(n)$ where $n$ is the number of keys stored
- An insert takes amortised expected $O(1)$ time
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic

where any two keys \( x, y \) are independent

We can define a stronger hash families with \( k \)-independence.

here the hash values of any choice of \( k \) keys are independent.

With \( k = \log n \) it is feasible to construct a \( k \)-independent family of hash functions

such that \( h(x) \) can be computed in \( O(1) \) time

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \( O(1) \) worst-case time,
- The space is \( O(n) \) where \( n \) is the number of keys stored
- An insert takes amortised expected \( O(1) \) time
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic

\[ k = \log n \]

where any two keys \( x, y \) are independent.

We can define a stronger hash families with \( k \)-independence.

here the hash values of any choice of \( k \) keys are independent.

With \( k = \log n \) it is feasible to construct a \( k \)-independent family of hash functions

such that \( h(x) \) can be computed in \( O(1) \) time

By changing the cuckoo hashing algorithm to perform a rehash after \( k = \log n \) moves

it can be shown (via a similar but harder proof) that the results still hold

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \( O(1) \) worst-case time,
- The space is \( O(n) \) where \( n \) is the number of keys stored
- An insert takes *amortised expected* \( O(1) \) time